Synchronization and concurrency measures for distributed computations

Michel Raynal, Masaaki Mizuno, Mitchell L. Neilsen

▶ **To cite this version:**

Michel Raynal, Masaaki Mizuno, Mitchell L. Neilsen. Synchronization and concurrency measures for distributed computations. [Research Report] RR-1539, INRIA. 1991. inria-00075023

HAL Id: inria-00075023
https://inria.hal.science/inria-00075023

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

N° 1539

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

SYNCHRONIZATION AND CONCURRENCY MEASURES FOR DISTRIBUTED COMPUTATIONS

Michel RAYNAL
Masaaki MIZUNO
Mitchell L. NEILSEN

Octobre 1991

RR-1539

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Synchronization and Concurrency Measures for Distributed Computations

Michel Raynal

Publication Interne n° 610 - Octobre 1991 - 20 pages - Programme 1

IRISA
Campus de Beaulieu
35042 Rennes-Cédex, FRANCE

Masaaki Mizuno Mitchell L. Neilsen

Department of Computing and Information Sciences
Kansas State University
Manhattan, Kansas 66506

Abstract

Time and message complexities are the usual measures used to characterize distributed computations. However, these measures only address quantitative aspects of the computation. Very little work has been devoted to defining more qualitative measures, such as the parallelism and the concurrency inherent in such computations.

This paper presents some qualitative measures from a synchronization point of view. These measures are based on the set of events that participate in the production of a given event (or of the whole computation). Such an approach allows us to precisely characterize the synchronization constraints inherent within a distributed computation, without being bothered by the perturbations caused by a given implementation. First, we introduce two abstractions, called cone and cylinder, associated with an event and a whole distributed computation, respectively. Then, the proposed concurrency measures are based on the analysis of these two abstractions.

A simple way to compute the measures at run-time is proposed. This implementation relies on two types of vector clocks that memorize the history of the computation, including the events produced and the synchronization delays. These measures can be very easily included in any system whose aim is to analyze distributed executions.

MESURES DE LA CONCURRENCE ET DE LA SYNCHRONISATION POUR LES EXECUTIONS REPARTIES.

M. Raynal, M. Mizuno, M.L. Neilsen

Les complexites en temps et en nombre de messages sont les 2 mesures habituellement utilisees pour caracteriser les executions reparties. L' inconnient de ces mesures est de se limiter a des aspects purement quantitatifs, Tres peu de travaux ont en effet aborde la definition de mesures plus qualitatives prenant en compte la concurrence ou le parallelisme inherent a une execution repartie.

Cet article presente des mesures qualitatives du point de vue de la synchronisation. Ces mesures sont basees sur l' ensemble des evenements qui participent a la production d' un evenement donne. Une telle approche permet de caracteriser la synchronisation presente dans un calcul reparti, sans etre gene par les perturbations possibles dues a l' implementation. Pour cela deux abstractions sont introduites associees respectivement a un evenement et au calcul en entier: le cone et le cylindre. Les mesures sont ensuite definies a l aide de ces abstractions.

Une mise en oeuvre simple de ces mesures, a l'execution, est proposee. Celle-ci s'appuie sur 2 types de vecteurs d' horloges logiques qui memorisent l' histoire du calcul et prennent en compte a la fois les evenements produits et les delais dus a la synchronisation.

Keywords: Asynchronous model, causality, concurrency, distributed computations, distributed systems, logical clock, measures, synchronization delay, vector clock.

1 Introduction

Time and space complexities are the quantitative measures generally used to characterize the efficiency of sequential algorithms and programs. In the field of parallel applications, numerical computations in particular, the notion of "speed-up" has been introduced to obtain some measures about performance gain obtained by the parallelism with respect to the best possible sequential algorithm which solves the same problem; such a measure takes into account the number of processors running the parallel program [2].

In the field of distributed algorithms and programs (by distributed we mean that there is no global memory and that all interactions between processes are by message exchange), the maximal number of messages and the computation time are the two measures usually encountered in the literature to characterize efficiency. Although these measures are interesting, they are quantitative and do not answer more qualitative questions such as whether the execution is well distributed and whether the execution has many waiting delays due to synchronization constraints. The aim of this paper is to present concurrency measures which answer these questions and are easily computed and well-suited for distributed computations. These measures, along with traditional quantitative measures such as message complexities, provide a better characterization of a distributed computation.

The organization of the paper is as follows: Section 2 provides an overview of the model of distributed computation, called an *asynchronous model*, which we use as the basis of our concurrency measures. Our concurrency measures are presented in Section 3. These measures are based on very simple notions, namely the causality cone associated with an event and the cylinder associated with an entire computation. Section 4 shows how these measures are computed by using logical clocks and vector clocks managed by processes and piggybacked by messages. Other additional measures are described in Section 5; they allow us to refine and to complete the preceding measures. Finally, Section 6 presents some other measures which have been proposed by other authors to answer the same types of questions.

2 Distributed Computation

2.1 Model of computation

A distributed program consists of a set of N processes $\{P_1, P_2, \cdots, P_N\}$ which cooperate to achieve a common goal. There is no global physical clock; that is, the system model is asynchronous. Processes must exchange messages to communicate and synchronize with each other. The underlying message passing system provides asynchronous communication and is assumed to be reliable; that is, there is no message loss or duplication, and no erroneous modification of messages. Communication delays are finite, but unpredictable.

Each process performs a sequence of actions which are modeled as a sequence of totally ordered events. The definition of an event is generally bound to the granularity of the actions it models. Since we are not interested in defining such granularity, we use a very broad definition of an event. Events are classified into the following three types:

1. *internal event:* Such an event models the execution of an action or a sequence of actions; these actions must not include send or receive operations.

2. *send event:* Such an event models a sequence of actions beginning or ending with a send operation; these actions must not include a receive operation. In other words, a send event is associated with the execution of each send operation.

3. *receive event:* Such an event models a sequence of actions beginning or ending with a receive operation, and not including a send operation. As with a send event, a receive event is associated with the execution of each receive operation.

As we can see, send and receive operations define "cutting points" that always separate events. The definitions of internal events and the "size" of the events in terms of the number of operations executed are of no concern here; this definition allows for more flexibility. At the lowest level, the execution of each operation can be modeled by an event.

Lamport has shown that such distributed asynchronous executions can be characterized by a partial order relation on the events produced; this relation called the *causality relation* (or *happened before relation)* and denoted by $<$, is defined in the following way [8]. Let E be a set of events produced by the execution of a distributed program. For $e, e' \in E$, $e < e'$ holds if

1. e and e' are events in the same process and e precedes e',

2. e is a send event and e' is the corresponding receive event, or

3. there exists e'' such that $e < e''$ and $e'' < e'$.

Two events e and e' are said to be *causally related* if $e < e'$ or $e' < e$ holds. If neither $e < e'$ nor $e' < e$ holds, these events are *concurrent* (or *mutually independent*), denoted by $e \parallel e'$; that is, $\neg((e < e') \vee (e' < e))$. Let $e_1, e_2, \cdots, e_k$ be a sequence of events such that $e_i < e_{i+1}$ for $1 \leq i \leq k-1$. Such a sequence is called a *causal path* from event e_1 to event e_k. Since the details of each event are unknown, we consider that all the events e such that $e < e'$ are necessary to produce e'. A computation, C, is a set of partially ordered events produced by N cooperating processes.

A computation may be graphically displayed in a *space-time diagram*, such as the one shown in Figure 1. In the figure, $e_{1,2}, e_{1,4}, e_{2,3}, e_{3,6}$, and $e_{3,7}$ are send events, and $e_{3,2}, e_{2,5}, e_{3,5}, e_{1,3}$, and $e_{1,5}$ are the associated receive events, respectively. Other events are internal events. There are many events which are concurrent. For example, events $e_{1,1}$ and $e_{2,1}$ are concurrent events. Events $e_{2,2}$ and $e_{1,4}$ are causally related, and the longest causal path from $e_{2,2}$ to $e_{1,4}$ is $e_{2,2}, e_{2,3}, e_{3,5}, e_{3,6}, e_{1,3}, e_{1,4}$.

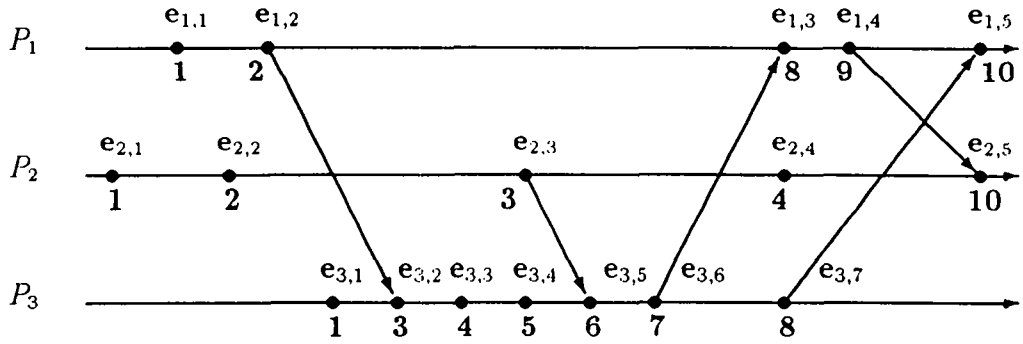

Figure 1. Computation C_1

2.2 Lamport's logical clock

Lamport presented a function c, called a *logical clock*, which maps a set of events E to a partially ordered set T [8]. Formally, $c : E \rightarrow T$ such that $e < e' \Rightarrow c(e) < c(e')$. Assume that T is the set of natural numbers, which is a partially ordered set. Then, the logical clock may be implemented by a set of counters, each of which is maintained by a different process in the system. Let c_i denote a counter maintained by process P_i. Each process P_i performs the following protocol:

1. When P_i executes an internal event, the clock value c_i is advanced by setting $c_i := c_i + d\,(d > 0)$.

2. When P_i executes a send event, the clock value c_i is advanced by setting $c_i := c_i + d (d > 0)$. The message carries the updated c_i value as the timestamp.

3. When P_i execute a receive event, where a message contains timestamp ts, the clock is advanced by setting $c_i := max\ (c_i, ts) + d (d > 0)$.

The value d represents the duration of the corresponding operation. In Figure 1, the number attached to each event represents the associated logical clock value c_i, assuming $d = 1$.

Note that this logical clock mechanism implicitly assumes that there are no delays when a message is transmitted. Logical time progresses as though message passing takes place instantaneously. It is possible to modify the third rule in the above protocol to account for logical communication delays (measured in logical time units). Let d_c be the communication delay, then the clock is advanced in rule 3 by setting $c_i := max\ (c_i, ts + d_c) + d$.

Let c_e be the logical time value associated with any event e, and let $d = 1$. Then, $(c_e - 1)$ events precede e on the longest causal path ending at e; this number is called the height of event e, denoted **height**(e). This means that no less than $(c_e - 1)$ events have been executed sequentially before event e, regardless of the processes that have produced these events. This notion of height will be used in Section 4.3, when we discuss synchronization and concurrency measures. For example, in Figure 1, since the logical clock value associated with event $e_{1,3}$ is 8, there are no less than 7 events which have been executed sequentially before the event.

3 Concurrency Measures

Any analysis technique should be independent of real time effects, such as system load and processor speed. To be independent of any particular machine executing the distributed program, we assume that message passing is instantaneous and that each event consumes approximately the same amount of computing time, called **time unit**. Since the definition of events is flexible, the later assumption may be achieved by adjusting the granularity of events. Such an approach allows us to precisely characterize the synchronization constraints inherent within a distributed computation, without being bothered by the perturbations caused by a given implementation.

In this model, the execution time, based on time units, may be computed using logical time described in Section 2.2 by assigning 1 to d. In the rest of the paper, the "logical time" of event e refers to the value c_e. Furthermore, we draw a space-time diagram by aligning all events which are given the same logical time

in the same column, as shown in Figure 2. Then, an event diagram for a given computation C is uniquely drawn.

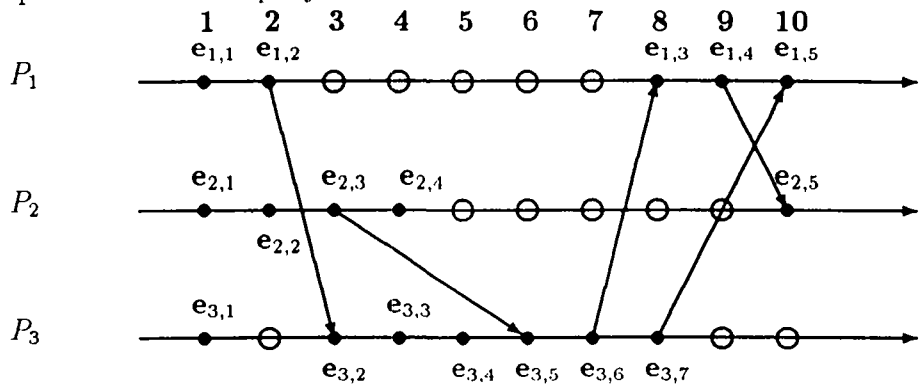

Figure 2. Computation C_1

3.1 Synchronization delay

The purpose of our concurrency measures is to quantify the total amount of synchronization delay within a computation. Consider computation C_1 shown in Figure 2. Process P_1 executes an internal event at logical time 1 ($e_{1,1}$) and then a send event at logical time 2 ($e_{1,1}$). The next event that the process P_1 executes is a receive event, $e_{1,3}$. However, since the corresponding send event $e_{3,6}$ does not occur until logical time 7, $e_{1,3}$ occurs at logical time 8. During the period between logical time 3 and logical time 7, Process P_1 has no internal or send event to perform and simply waits to receive the message from Process P_3. Thus, we may conclude that process P_1 has wasted 5 time units due to synchronization delay, and that the degree of concurrency is decreased. In Figure 2, synchronization delay is represented by circles, where events are not produced by processes.

If some of the internal events may be moved from other processes (Process P_3, in particular, in this case) to process P_1, more concurrency may be achieved and the total time to execute the same computation may be shortened. Thus, the total amount of synchronization delay in a computation gives a good measure of concurrency. Note that Process P_3 finishes its last event, $e_{3,7}$, at logical time 8 and has nothing to do until computation C_1 completes at logical time 10. The time units corresponding to this waiting period is also considered to be synchronization delay.

Recall that the message passing system provides asynchronous communication. Thus, a sending process is never blocked. For example, Process P_2 executes send event $e_{2,3}$ at logical time 3. Process P_3 does not executes the corresponding receive event $e_{2,3}$ until logical time 6. However, Process P_2 is not blocked.

We present two types of concurrency measures $\alpha_\epsilon(e)$ and $\alpha(C)$. Concurrency

measure $\alpha_e(e)$ may be computed for each event e, and indicates how many time units are waisted by synchronization delay to produce event e. Concurrency measure $\alpha(C)$ is computed for a whole computation C (from the beginning of the execution until the completion of the last event). This measure indicates how many time units are waisted due to synchronization delay during the entire computation of C.

3.2 Cone and cylinder abstractions

In order to quantify synchronization delay, we first introduce two abstractions, called **CONE** and **CYLINDER**. They represent the "shape" of a computation in part (CONE) or in whole (CYLINDER) from the point of view of synchronization delay. A CONE is associated with an event; the cone associated with an event e represents the partially ordered set of all events that causally precede event e. Thus, our CONE abstaction is the dual of the cone abstraction presented by Lamport [9]. For example, in Figure 2, CONE($e_{2,5}$) represents the partially ordered set of events consisting of $e_{1,1}$, $e_{1,2}$, $e_{1,3}$, $e_{1,4}$, $e_{2,1}$, $e_{2,2}$, $e_{2,3}$, $e_{2,4}$, $e_{3,1}$, $e_{3,2}$, $e_{3,3}$, $e_{3,4}$, $e_{3,5}$, and $e_{3,6}$. A CYLINDER is associated with a whole execution C; it represents the partially ordered set of events produced by this computation. Thus, in the entire computation C_1 represented by Figure 2, CYLINDER(C_1) consists of all the events in the figure. With each of these abstractions, three values are associated: **volume**, **weight**, and **height**.

The weight of CONE(e), denoted by **weight**(CONE(e)), is the exact number of events which actually do causally precede e. The weight of CYLINDER(C), denoted by **weight**(CYLINDER(C)), is the number of events which are actually produced in the execution of C. In Figure 2, **weight**(CONE($e_{2,5}$)) = 14, and **weight** (CYLINDER (C_1)) = 17.

The volume of CONE(e), denoted **volume** (CONE(e)), represents the maximum number of events that could possibly causally precede e. Similarly, **volume** (CYLINDER(C)) represents the maximum number of events that could be produced during the whole execution C. For example, in Figure 2, **volume**(CONE($e_{2,5}$)) = 25. This value is easily obtained in the figure by counting all the events and synchronization delay (denoted by circles) in the area associated with CONE($e_{2,5}$). Similarly, **volume**(CYLINDER(C_1)) = 30.

The height of CONE(e), denoted by **height** (CONE(e)), represents the the total number of events which causally precede e, on the longest causal path ending in e. The height of CYLINDER(C), denoted by **height** (CYLINDER(C)), represents the the largest logical time associated with an event in CYLINDER(C). In Figure 2, **height**(CONE($e_{2,5}$)) = 9, and **height**(CYLINDER(C_1)) = 10.

3.3 Concurrency measures

The total number of time units waisted by synchronization delay will be the difference between the **volume** and **weight** of the abstraction. This leads to the following concurrency measures:

$$\alpha'_e(e) := \frac{\textbf{volume}(\text{CONE}(e)) - \textbf{weight}(\text{CONE}(e))}{\textbf{volume}(\text{CONE}(e)) - \textbf{height}(e)}$$

$$\alpha'(C) := \frac{\textbf{volume}(\text{CYLINDER}(C)) - \textbf{weight}(\text{CYLINDER}(C))}{\textbf{volume}(\text{CYLINDER}(C)) - \textbf{height}(C)}$$

In both of the above equations, the numerator denotes the total synchronization delay which actually occurred. The denominator denotes the maximum synchronization delay theoretically possible in the computation, and is used for normalization. The above equations are not defined when the denominator is zero. This situation occurs when only one process is involved in a computation; that is, the measures are only defined for communicating programs.

Define $\alpha_e(e) := 1 - \alpha'_e(e)$ and $\alpha(C) := 1 - \alpha'(C)$. The reason for subtracting the fraction from 1 is to make the measures compatible with other concurrency measures; that is, $\alpha = 1$ stands for a maximally concurrent computation and $\alpha = 0$ stands for a totally sequential computation. In Figure 2, $\alpha_e(e_{2,5}) = 0.31$, and $\alpha(C_1) = 0.35$. Note that α_e values are not defined for events $e_{1,1}$, $e_{1,2}$, $e_{2,1}$, $e_{2,2}$, $e_{2,3}$, $e_{2,4}$, and $e_{3,1}$.

4 Computation of Concurrency Measures

4.1 Computation of weight

Vector time, independently introduced by Fidge and Mattern [6, 11], may be used to compute the number of events which have actually preceded event e. The system allocates a vector of N counters, V_i, to each process. Let $V_i[j]$ denote the jth counter maintained by process P_i. All values in V_i are initialized to zero. Each process P_i follows the following protocol:

1. When P_i executes an internal event, V_i is advanced by setting $V_i[i] := V_i[i] + 1$.

2. When P_i executes a send event, V_i is advanced by setting $V_i[i] := V_i[i] + 1$. The message carries the updated V_i value.

3. When P_i execute a receive event, where a message contains V_j, V_i is advanced by setting $V_i[k] := \max(V_i[k], V_j[k])$ for $1 \leq k \leq N$, and then $V_i[i] := V_i[i]+1$.

Let V_i^e denote values of vector V_i at the time event e occurs at process P_i. Let V_i^C denote values of V_i when the computation C terminates. Then, $V_i^e[j]$ represents the number of events which occur at process P_j and causally precede event e. The values of V_i for the computation C_1 in Figure 2, are given below in Figure 3. For example, $V_2^{e_{2,5}} = (4, 5, 6)$; that is, $V_2^{e_{2,5}}[1] = 4$, $V_2^{e_{2,5}}[2] = 5$, and $V_2^{e_{2,5}}[3] = 6$. Thus, value **weight**(CONE(e)), the exact number of event which causally affected event e, is computed by

$$\textbf{weight}(\text{CONE}(e)) := (\textstyle\sum_{j=1}^{N} V_i^e[j]) - 1.$$

The reason for subtracting 1 is to exclude event e from the count.

After the execution of C terminates, **weight**(CYLINDER(C)), the exact number of event which actually occurred in computation C, is computed by

$$\textbf{weight}(\text{CYLINDER}(C)) := \textstyle\sum_{j=1}^{N} V_j^C[j].$$

4.2 Computation of volume

For an event e occuring at process P_i, **volume**(CONE(e)) may be computed by storing the logical clock value of the last event at every other process which causally affected event e at process P_i. Such values may be obtained by extending the implementation of logical clocks presented in the previous section.

Instead of allocating a single counter c_i, a vector of N counters W_i is allocated to each process P_i. Counter $W_i[i]$ maintains the c_i value, and $W_i[j], i \neq j$, stores the c_j value of the last send event at process P_j which logically precedes the event that occurred at process P_i at logical time $W_i[i]$. All values in W_i are initialized to zero. Each process P_i follows the following protocol:

1. When P_i executes an internal event, W_i is advanced by setting $W_i[i] := W_i[i] + 1$.

2. When P_i executes a send event, W_i is advanced by setting $W_i[i] := W_i[i] + 1$. The message carries the updated W_i value.

3. When P_i execute a receive event, where a message contains W_j ($W_j[j]$ is the logical time of the corresponding send event at process P_j), then W_i is advanced as follows:

   (a) $W_i[k] := \max(W_i[k], W_j[k])$ for $1 \leq k \leq N$.

   (b) $W_i[i] := \max(W_i[i], W_j[j]) + 1$.

Let W_i^e denote values of vector W_i at the time event e occurs. The values of W_i for the events in the computation C_1 in Figure 2, are given below in Figure 3. For example, $W_2^{e_{2,5}} = [9, 10, 7]$.

Then, value **volume**(CONE(e)), the maximum number of events which could possibly precede event e, is computed by

volume(CONE(e)) := $(\sum_{1 \leq j \leq N} W_i^e[j]) - 1$.

Let W_i^C denote values of W_i when the computation C terminates. The maximum number of events which could possibly occur during the entire computation of C, **volume**(CYLINDER(C)), is computed after the computation by

volume(CYLINDER(C)) = $N *$ **height**(CYLINDER(C)).

4.3 Computation of height

The logical clock value associated with event e at process P_i, denoted $W_i^e[i]$, represents the total number of events which precede event e, including event e in the computation. In other words, the clock value represents the number of events on the longest causal path from any event to event e. Thus, **height**(CONE(e)) is computed by

height(CONE(e)) = $W_i^e[i] - 1$.

The height of CYLINDER(C) is computed by

height(CYLINDER(C)) = $\max_{1 \leq j \leq N} W_j^C[j]$.

4.4 Computation of concurrency measures

Concurrency measures α_e and α are computed by using the vectors W and V as follows:

$$\alpha_e(e) := 1 - \frac{\sum_{j=1}^{N} W_i^e[j] - \sum_{j=1}^{N} V_i^e[j]}{\sum_{j=1}^{N} W_i^e[j] - W_i^e[i]}$$

$$\alpha(C) := 1 - \frac{N * \max_{1 \leq j \leq N} W_j^C[j] - \sum_{j=1}^{N} V_j^C[j]}{(N-1) * \max_{1 \leq j \leq N} W_j^C[j]}$$

Note that $\alpha_e(e)$ may be computed for each event e by process P_i at run time by using only local data structures maintained by the process. However, evaluation of $\alpha(C)$ requires information stored in all of the processes involved in the computation C. Thus, the value may be obtained after the computation, by collecting information from all N processes.

4.5 Examples

This subsection presents some examples of concurrency measures. Figure 3 shows vector values V_i and W_i, at each process P_i for $1 \leq i \leq 3$, associated with each event in the computation represented by Figure 2. The values of V_i and W_i are represented by $(\cdots)$ and $[\cdots]$, respectively. Since $V_2^{e_{2,5}} = (4, 5, 6)$ and

$W_2^{e_{2,5}} = [9, 10, 7]$, $\alpha_e(e_{2,5}) = 1 - \frac{(9+10+7)-(4+5+6)}{(9+10+7)-10} = 0.3125$. Similarly, $\alpha(C_1) = 1 - \frac{3*10-(5+5+7)}{2*10} = 0.35$.

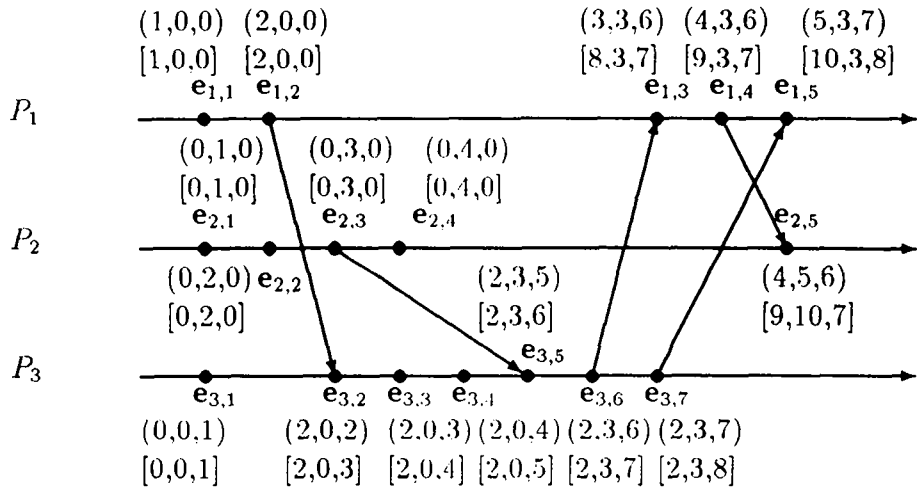

Figure 3. Computation C_1

Figure 4 shows a totally sequential computation. The α_e values at $e_{2,1}$, $e_{2,2}$, $e_{2,3}$, $e_{3,1}$, $e_{3,2}$, and $e_{3,3}$ are all 0. The $\alpha(C_2)$ value is also 0. Note that the α_e values at $e_{1,1}$ and $e_{1,2}$ are not defined.

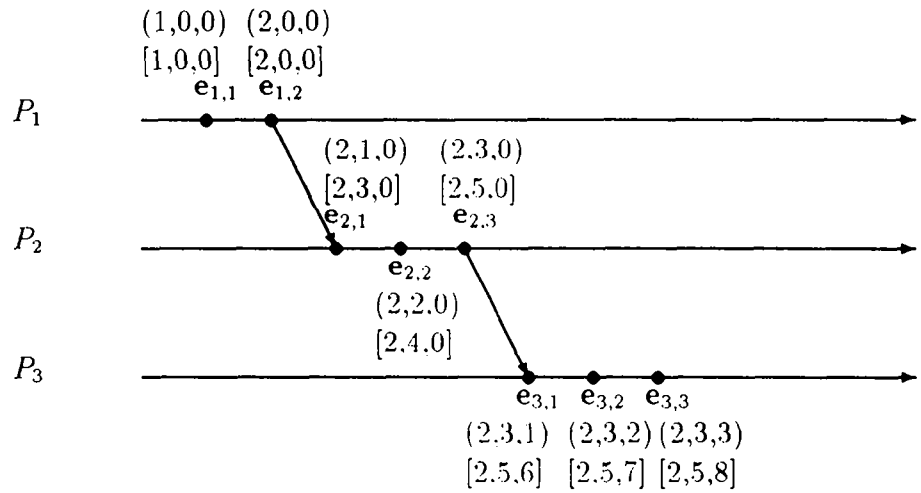

Figure 4. Computation C_2

Figure 5 shows a complete concurrent execution. The α_e values at events $e_{1,3}$, $e_{1,4}$, $e_{2,2}$, $e_{2,3}$, and $e_{2,4}$ are all 1. The $\alpha(C_3)$ value is also 1.

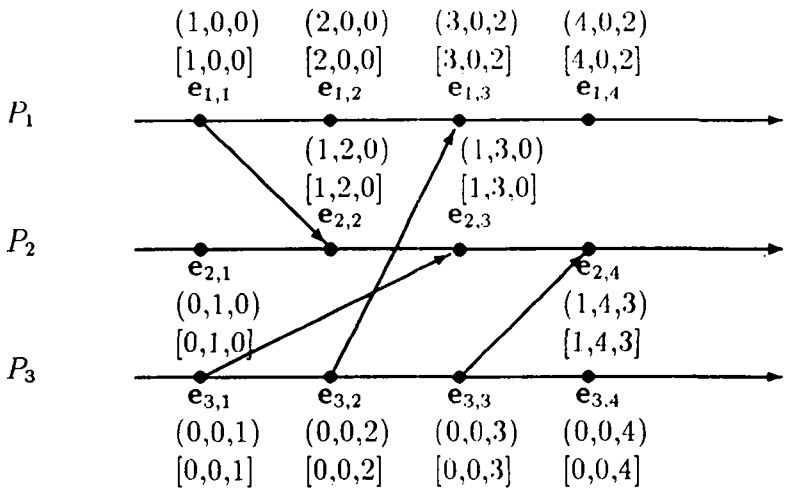

Figure 5. Computation C_3

5 Additional Measures

By using logical clock vectors W and V, several other useful measures may be computed.

- $\theta_1(e,i,j) = W_i^e[j] - V_i^e[j]$: This value is the amount of synchronization delay at process P_j, when working to produce event e at process P_i.
- $\theta_1'(e,i,j) = \frac{W_i^e[j]-V_i^e[j]}{W_i^e[j]}$: This value is the percentage of logical time lost due to synchronization delay at process P_j to produce event e at process P_i.
- $\theta_2(e,i,j) = \frac{V_i^e[j]}{W_i^e[i]}$: This value is the percentage of the events produced by process P_j which causally precede event e at process P_i relative to the logical time taken to produce event e.
- $\theta_3(e,i,j) = \frac{V_i^e[j]}{\sum_{1\le k\le N} V_i^e[k]}$: This value is the percentage of actual computation done by process P_j, relative to the total computation done by all the processes, to produce event e at process P_i.

The above measures are associated with an event e. The same measures may be applied to the whole computation C as follows:

- $\theta_4(C,j) = (\max_{1\le k\le N} W_k^C[k]) - V_j^C[j]$: This value is the amount of synchronization delay at process P_j in computation C.
- $\theta_4'(C,j) = \frac{(\max_{1\le k\le N} W_k^C[k])-V_j^C[j]}{\max_{1\le k\le N} W_k^C[k]}$: This value is the percentage of logical time lost due to synchronization delay at process P_j relative to the logical time taken to complete computation C.

- $\theta_5(C,j) = \frac{V_j^C[j]}{\max_{1\leq k\leq N} W_k^C[k]}$: This value is the percentage of the events produced by process P_j relative to the logical time taken to complete computation C.
- $\theta_6(C,j) = \frac{V_j^C[j]}{\sum_{1\leq k\leq N} V_k^e[k]}$: This value is the percentage of actual computation done by process P_j relative to the whole computation of necessary to complete C. This value shows the distribution balance of load among processes.

Consider computation C_1 and event $e_{2,5}$ in Figure 3. Since $W_2^{e_{2,5}} = [9, 10, 7], V_2^{e_{2,5}} = (4, 5, 6)$, the above measures for event $e_{2,5}$ give the following values:

$\theta_1(e_{2,5}, 2, 1) = 5$	$\theta_1(e_{2,5}, 2, 2) = 5$	$\theta_1(e_{2,5}, 2, 3) = 1$
$\theta_2(e_{2,5}, 2, 1) = 0.4$	$\theta_2(e_{2,5}, 2, 2) = 0.5$	$\theta_2(e_{2,5}, 2, 3) = 0.6$
$\theta_3(e_{2,5}, 2, 1) = 0.27$	$\theta_3(e_{2,5}, 2, 2) = 0.33$	$\theta_3(e_{2,5}, 2, 3) = 0.4$

Similarly, the above measures for computation C_1 give the following values:

$\theta_4(C_1, 1) = 5$	$\theta_4(C_1, 2) = 5$	$\theta_4(C_1, 3) = 3$
$\theta_5(C_1, 1) = 0.5$	$\theta_5(C_1, 2) = 0.5$	$\theta_5(C_1, 3) = 0.7$
$\theta_6(C_1, 1) = 0.29$	$\theta_6(C_1, 2) = 0.29$	$\theta_6(C_1, 3) = 0.41$

These additional measures can be used to obtain a better understanding of a given distributed computation. Combined with the other concurrency measures, they allow us to more precisely characterize the profile of a computation. Other additional measures using vectors V and W may be envisaged.

6 Comparison of Concurrency Measures

This section briefly review two existing concurrency measures: Charron-Bost's and Fidge's concurrency measures. Then, we compare them with α.

6.1 Charron-Bost's measure

Charron-Bost's concurrency measure is based on the principle that how often the stopping of one process would block other processes; that is, blocking less often implies that the computation is more concurrent [4]. This measure is closely related to the number of possible consistent cuts in a given computation. A *consistent cut* CT of a distributed computation C, consisting of a partially ordered set of events E, is a finite subset CT $\subseteq E$ such that $e \in$ CT and $e' < e \Rightarrow e' \in$ CT [3, 7].

The reasoning behind this concurrency measure is that the tolerance of a computation regarding the stopping of one process has a strong relation with its ability to be cut in a consistent way. Thus, the more consistent cuts a computation

has, the more concurrent the computation is. The concurrency measure m of a distributed computation C is defined by:

$$m(C) = \frac{\mu - \mu^s}{\mu^c - \mu^s}$$

where μ is the number of consistent cuts in computation C, and μ^s, μ^c are the number of consistent cuts in the sequential computation and in the entirely concurrent computation consisting of the same number of processes, each of which consists of the same number of events, respectively.

Concurrency measure m characterizes the degree of concurrency of a given computation C well. However, the computation of μ is not feasible. This is the major drawback. Furthermore, m is defined only for the entire computation of C, not for each event within C.

6.2 Fidge's measure

Fidge proposed another concurrency measure, called β, which is easy to compute and measures either event e or the entire computation C [5]. Concurrency measure β is defined by

$$\beta = \frac{\rho - \tau}{\rho - 1}$$

where ρ and τ denote, respectively, the minimum number of events that must occur before the current point in the computation, and the minimum logical time, in units of events, required to reach a particular point in the computation. The numerator denotes the amount of logical time "saved" by the use of concurrency, and the denominator denotes the time that could potentially be saved given unlimited computing resources.

The measures, $\beta(e)$ and $\beta(C)$, may be represented by using the abstractions CONE(e) and CYLINDER(C) as follows:

$$\beta(e) = \frac{\textbf{weight}(\text{CONE}(e)) - \textbf{height}(\text{CONE}(e))}{\textbf{weight}(\text{CONE}(e)) - 1}$$

$$\beta(C) = \frac{\textbf{weight}(\text{CYLINDER}(C)) - \textbf{height}(\text{CYLINDER}(C))}{\textbf{weight}(\text{CYLINDER}(C)) - 1}$$

In fact, the numerator of $\beta(C)$ is the same as that of $\alpha(C)$; the difference is in the denominators. However, $\beta(e)$ is significantly different from $\alpha_e(e)$. Consider examples shown in Figures 6 and 7.

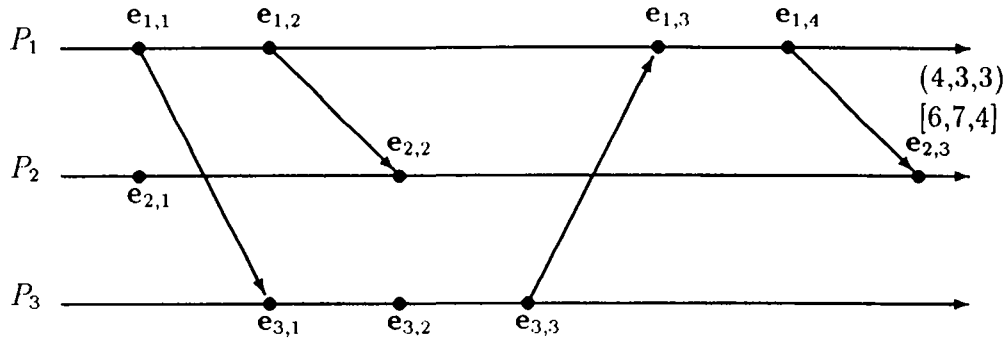

Figure 6. Computation C_4

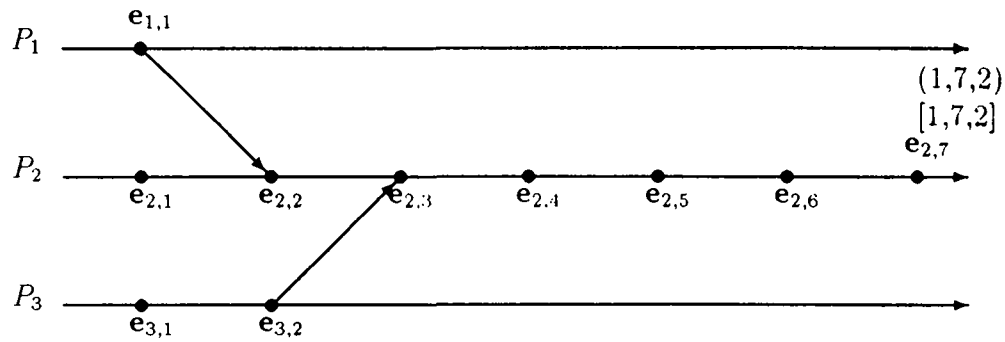

Figure 7. Computation C_5

The weight and height of the cone associated with event $e_{2,3}$ in computation C_4 and event $e_{2,7}$ in computation C_5 are the same. However, α_e and β values for these events are different as follows:

$\alpha_e(e_{2,3}) = 0.3$ $\qquad$ $\alpha_e(e_{2,7}) = 1$
$\beta(e_{2,3}) = 0.33$ $\qquad$ $\beta(e_{2,7}) = 0.33$

Fidge's measure, β, gives the same value for both events $e_{2,3}$ in C_4 and $e_{2,7}$ in C_5. This is because in both computations, there are exactly three events which causally precede the events, besides the events on the longest causal path. However, α_e gives different values for these events. In computation C_5, $\alpha_e(e_{2,7}) = 1$ since there is no synchronization delay in CONE($e_{2,7}$). On the other hand, the value, $\alpha_e(e_{2,3}) = 0.3$, indicates that there is not much concurrency used to produce event $e_{2,3}$. In fact, the volume of CONE($e_{2,3}$) is much larger than that of CONE($e_{2,7}$). This suggests that much less total computation time is spent to produce event $e_{2,7}$ than to produce $e_{2,3}$ since in computation C_4, much time is waisted for synchronization delay, whereas in computation C_5, no computation time is waisted

for synchronization delay. Thus, more computations could potentially be done in C_5 than in C_4. For example, in computation C_5, eleven more events may be potentially produced by Processes P_1 and P_3, outside of CONE($e_{2,7}$). However, in computation C_4, only four more events may be produced outside of CONE($e_{2,3}$).

The idea behind our measure is that if α_e is maximized for each event, then the concurrency of a whole computation will be maximized. The only case in which this rule does not apply is when distribution of computational load is unbalanced; that is, some processes finish their jobs earlier than others and wait for the completion of the whole computation. This balance is measured by $\theta_6(C, j)$ described in Section 5. Furthermore, if α_e is computed for each event, these values are useful to "tune up" the computation since they give information about what parts of the computation have much synchronization delay. This feature is particularly important if the measure is used in certain applications, such as interactive monitoring or debugging.

7 Conclusion

Time and message complexities are the usual measures used to characterize distributed computations. However, these measures only address quantitative aspects of the computation. Very little work has been devoted to defining more qualitative measures, such as the parallelism and the concurrency inherent in such computations. Currently, research on optimizing parallelism in distributed systems is an active area of research [1, 10].

This paper has presented some qualitative measures from a synchronization point of view. These measures are based on the set of events that participate in the production of a given event (or of the whole computation). Such an approach allows us to precisely characterize the synchronization constraints inherent within a distributed computation, without being bothered by the perturbations caused by a given implementation. Thus, these measures are independent of the underlying system running the computation, and consequently they characterize exactly the synchronization (through the measures of the delays it involves) inherent in a distributed computation. The definition of these measures has been based on two well defined abstractions, namely cone and cylinder, to which simple measures can be associated: volume, weight, and height. The proposed concurrency measures come from the analysis of these two abstractions.

A simple way to compute the measures at run-time has been proposed. This implementation relies on two types of vector clocks that memorize the history of the computation, including the events produced and synchronization delay.

These two measures (with the additional measures) can be very easily included in a any system whose aim is to analyze distributed executions (distributed debugging, monitoring, etc). Such measures can allow the system to improve the

sharing of computational power when time consuming applications, such as image and signal processing, are implemented on distributed memory parallel machines.

References

[1] O. Berry and D. Jefferson. Critical path analysis of distributed simulation. In *Proceedings of the Conference on Distributed Simulation*, pages 57–60, 1985.

[2] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation, Numerical Methods.* Prentice-Hall, Inc., 1989.

[3] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Programming Languages and Systems*, 3(1):63–75, 1985.

[4] B. Charron-Bost. Combinatorics and geometry of consistent cuts: Applications to concurrency theory. In Bermond and Raynal, editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 45–56, Nice, France, 1989. Springer-Verlag, LNCS 392.

[5] C.J. Fidge. A simple run-time concurrency measure. In *Proceedings of the 3rd Australian Transputer and OCCAM User Group Conference*, pages 92–101, 1990.

[6] C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[7] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, 1987.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[9] L. Lamport. The mutual exclusion problem: part i - a theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.

[10] M. Livny. A study of parallelism in distributed simulation. In *Proceedings of the Conference on Distributed Simulation*, pages 94–98, 1985.

[11] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robrt, editors, *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Bonas, France, 1989. North-Holland.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 604 A GENERAL METHOD TO DEFINE QUORUMS
Mitchell L. NEILSEN, Masaaki MIZUNO, Michel RAYNAL
Septembre 1991, 20 pages.

PI 605 OBTENTION DES EQUATIONS DYNAMIQUES D'UN SYSTEME PHYSIQUE A PARTIR DE SON MODELE BOND GRAPH
Bénédicte EDIBE
Septembre 1991, 26 pages.

PI 606 CONSTRUCTIVE PROBABILITY AND THE SIGNalea LANGUAGE : BUILDING AND HANDLING RANDOM PROCESSES VIA PROGRAMMING
Albert BENVENISTE
Septembre 1991, 60 pages.

PI 607 ABOUT LOGICAL CLOCKS FOR DISTRIBUTED SYSTEMS
Michel RAYNAL
Octobre 1991, 16 pages.

PI 608 UNE NOUVELLE APPROCHE REALISTE DE SIMULATION D'ECLAIRAGE DANS UN ENVIRONNEMENT DIFFUS
Eric LANGUENOU, Kadi BOUATOUCH, Pierre TELLIER
Octobre 1991, 64 pages.

PI 609 INTEGRATION D'UN CORRECTEUR ORTHOGRAPHIQUE DANS L'EDITEUR STRUCTURE GRIF
Patrice FRISON, Eric PICHERAL, Hélène RICHY
Octobre 1991, 22 pages.

PI 610 SYNCHRONIZATION AND CONCURRENCY MEASURES FOR DISTRIBUTED COMPUTATIONS
Michel RAYNAL
Octobre 1991, 20 pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399